BTA Algorithms to Ensure Termination of Off-line Partial Evaluation

Arne J. Glenstrup and Neil D. Jones, DIKU, University of Copenhagen
e-mail: {panic,neil}@diku.dk

Abstract. A partial evaluator, given a program and a known "static" part of its input data, outputs a residual program in which computations depending only on the static data have been precomputed [3]. The ideal is a "black box" which discovers and performs nontrivial static computations whenever possible, and never fails to terminate. Practical partial evaluators fall short of this goal: they sometimes loop (typical of functional programing partial evaluation), or terminate but are excessively conservative (typical in partial deduction[1]). This paper presents efficient algorithms (being implemented) for binding-time analysis for off-line specialisers. They ensure that the specialiser performs many nontrivial static computations, and are at the same time guaranteed to terminate.

1 Introduction

Henceforth `p` will be a program to be specialised, some of whose inputs, called "static," are known at specialisation time; remaining inputs are unknown and called "dynamic." We assume every function parameter in program `p` is either *totally static* or *totally dynamic*. Thus there is no *partially static data* in our examples.

Focus. A broad and on the whole successful application area is using "off-line" specialisation tools to remove overhead from programs with a high interpretive overhead. A partial evaluator is analogous to a spirited horse: while impressive results can be obtained when used well, the user must know what he/she is doing. If not used with considerable finesse, current specialisers may

- loop infinitely (at *compile time*, quite bad for new users), or
- yield enormous programs, and may even
- pessimise rather than optimise.

Hand tuning `p` is usually allowed (and often necessary) to obtain efficient and small target programs when specialising an interpreter with respect to a known source program: most systems allow a "loophole" whereby parts can be hand marked, e.g. "make this dynamic" or "don't unfold this function call."

On-line and Off-line Specialisation. Individual unfolding and symbolic computation acts are simple and obviously semantics-preserving; the problem is controlling their repeated use. One strategy is *on-line* specialisation, in which `p`'s

[1] Recent reasearch has addressed this problem for on-line specialisation [11, 9].

execution is mimicked directly, and decisions are taken on the fly as to which of p's operations to suspend. The other, *off-line* specialisation, begins with a program analysis called *binding-time analysis*, henceforth BTA, in which it is only known *which* inputs will be available during specialisation but not *what* their values are. BTA decides *before specialisation* which operations or commands to simulate and which to suspend, resulting in an *annotated* program, as in Similix [6]. A familiar example ("_S" = static, "_D" = dynamic, "lift" = make a static value dynamic):

```
append(X_S,Y) = if_S X =_S nil then Y
                else cons_D(lift(hd_S X), append_unfold(tl_S X, Y))
```

In our experience BTA annotations give invaluable feedback for binding-time improvements; these will also improve results of on-line specialisation.

What's New? We develop algorithms for a more conservative BTA which is *guaranteed* to lead to termination at specialisation time and which, on interpreter examples familiar to us, are liberal enough to annotate all expected static operations as static. The algorithms use efficient manipulation of data flow graphs (depth-first search, strongly connected components, etc.), and will be added to a future release of Similix.

They recognize (in order) three properties: independence from dynamic input; being bounded by some static input; and "bounded anchorage," a reworking of Holst's *in situ* condition [4]. This paper sketches efficient algorithms for the first two, and relatively efficient algorithm for the last is being developed. The methods are powerful enough to establish termination by *discovering* lexicographical orderings on sequences of argument tuples, and simple examples show them even more general than this.

2 Background

Programs to be specialised are assumed in an *untyped first-order functional* language, with a usual syntax. We particularly wish to compile by specialising interpreters, so the example program is an interpreter. However the *languages that are interpreted* can be functional, imperative, higher-order, etc.

The Problem Source: the Over-Strict Nature of Specialisation. Most specialisers only do simple operations: unfolding, symbolic computation, and definition of new specialised variants of p's functions. The specialiser's job is to perform certain of p's operations or commands possible using only static inputs, and to *suspend*, i.e. to generate code for, the remaining ones [3].

The key to termination of the specialiser is to recognise when static parameters can take only finitely many different values during specialisation, thus being of *bounded static variation* or BSV, formally defined later.

When the specialiser encounters a conditional construct with dynamic test, it must specialise *both* branches (perhaps including "semantically dead code"). It is therefore *more strict* than CBV-evaluation. This over-strictness is the source of the termination problem: intuitively, a recursive function controlled by dynamic data can risk being specialised into infinitely many versions, even though normal evaluation of `p` on *any particular* input tuple would eventually reach the base case and terminate.

3 Programming Languages `F` and `TRF`

We consider a simple, first order call-by-value functional language, `F`:

$$\begin{array}{lll}
\texttt{p} : \textit{Program} & ::= & \textit{f1}\,\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_{m_1}\texttt{)} \texttt{ = } e_1 \ \ldots\ \textit{fn}\texttt{(}z_1\texttt{,}\ldots\texttt{,}z_{m_n}\texttt{)} \texttt{ = } e_n \\
e : \textit{Expression} & ::= & se \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
 & \mid & e_1 \texttt{ where } x \texttt{ = } e_2 \mid \textit{fi}\texttt{(}se_1\texttt{,}\ldots\texttt{,}se_{m_i}\texttt{)} \\
se : \textit{SimpleExpression} & ::= & be \mid \textit{basefcn}\texttt{(}be_1\texttt{,}\ldots\texttt{,}be_k\texttt{)} \\
be : \textit{BasicExpression} & ::= & x \mid \textit{constant}
\end{array}$$

For technical reasons nested calls are disallowed, but their effect can be achieved by `where` expressions. Further, we assume no formal parameter or `where` variable name is used twice. Without loss of generality we assume program input is passed via *f1*'s parameters, and that no calls to *f1* occur anywhere in `p`. For brevity our algorithms treat the *tail recursive* subset of `F`, but are easily extended to all of `F` (as in the examples). We let functions arity : *FunctionNames* $\rightarrow \mathbb{N}$ and body : *FunctionNames* $\rightarrow$ *Expressions* give a function's number of parameters and its defining expression, and let $\#$ be the length function on tuples. Further, $\overline{v}$ is shorthand for $\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)}$ and $\overline{v}_i$ is the ith component of a tuple.

Program Semantics (tail recursive fragment). A *state* is a pair $(f, \overline{v})$, with f defined in `p` and $\#v = \text{arity}(f)$. Given a value set V, we define *call-free evaluation* of expression e as

$$[\![e]\!]_0\overline{v} = \begin{cases} w, & \text{if } e\text{'s value } w \text{ on } \overline{v} \text{ is computable without function calls.} \\ (g, \overline{w}), & \text{if } e\text{'s value on } \overline{v} \text{ is } g\text{'s value on } \overline{w} \text{ (by an immediate call from} \\ & e \text{ to } g). \end{cases}$$

A single-step *state transition*, written $(f, \overline{v}) \rightarrow (g, \overline{w})$, occurs if `p` contains $f\,\overline{x}$ `=` e, and $[\![e]\!]_0\overline{v} = (g, \overline{w})$. Total evaluation of an expression can now be defined by $[\![e]\!]\overline{v} = v$ iff either $[\![e]\!]_0\overline{v} = v$ or $[\![e]\!]_0\overline{v} = (g, \overline{w})$ and $[\![\text{body}(g)]\!]\overline{w} = v$.

4 Off-line Specialisation

Given a binding-time annotated program, an off-line specialiser will specialise the program by symbolic evaluation, starting with the goal function (*f1*). It does so by *evaluating* static expressions and *generating residual code* for dynamic expressions. *Example*: Specialising `append` to static input `X` = `[1,2]` yields

```
append([1,2],Y) = if_S [1,2] =_S nil then Y else
                  cons_D(lift(hd_S[1,2]),append_unfold(tl_S[1,2],Y))
                = cons_D(lift(hd_S[1,2]),append_unfold(tl_S[1,2],Y))
                = cons_D(1,append_unfold([2],Y))
                = cons_D(1,if_S [2] =_S nil then Y else
                  cons_D(lift(hd_S[2]),append_unfold(tl_S[2],Y)))
                = ··· = cons(1,cons(2,Y))
```

A simple interpreter skeleton is shown in Figure 1; `case` is syntactic sugar and `list(x)` = `cons(x,'nil)`. Initial binding time division: `prog` is static and `input` is dynamic. Functions `look_body` and `look_name` return body and parameter lists, and `look_var(X, ns, vs)` returns the value in `vs` (*value list*) corresponding to `X`'s position in `ns` (*name list*).

```
Run(prog, input) ; Program × V → V
  = Eval(e1, list(n1), list(input), prog)
      where  e1 = look_body(first_fcn(prog), prog)
             n1 = look_name(first_fcn(prog), prog)
Eval(e, ns, vs, pgm) ; Expression × NameList × VList × Program → V
  = case e of
      constant          : constant
      X                 : look_var(X, ns, vs)
      (basefcn e1 e2)   : apply(basefcn, v1, v2)
                          where  v1 = Eval(e1, ns, vs, pgm)
                                 v2 = Eval(e2, ns, vs, pgm)
      (if e0 e1 e2)     : if Eval(e0, ns, vs, pgm)
                          then Eval(e1, ns, vs, pgm)
                          else Eval(e2, ns, vs, pgm)
      (call f e0)       : Eval(e3, list(n3), list(v3), pgm)
                          where  e3 = look_body(f, pgm)
                                 n3 = look_name(f, pgm)
                                 v3 = Eval(e0, ns, vs, pgm)
      (e1 where X=e2)   : Eval(e1, cons(X, ns), cons(v4, vs), pgm)
                          where  v4 = Eval(e2, ns, vs, pgm)
look_body(f, pg) ;  Name × FunctionDefList → FunctionBody
  = case car(pg) of
      (g(x) = e)        : if f = g then e else look_body(f, cdr(pg))
look_names(...)   =       — similar to look_body —
look_var(...)     =       — similar to look_body —
```

Figure 1: A Simple Interpreter

5 Bounded Static Variation, BSV

Let ';' be tuple concatenation and let *f1* be p's start function. Suppose $s + d =$ arity(*f1*), where the first s parameters of *f1* are static, and the remaining d dynamic. Intuitively, any parameter is BSV if for each value of static input data, $\overline{\sigma}$, the set of values it can take for arbitrary dynamic input data, $\overline{\delta}$, is finite.

Definition 1 (Bounded Static Variation, BSV). The ith parameter of f, f_i, is of *bounded static variation*, also written BSV(f_i), iff

$$\forall \overline{\sigma} \in V^s : \{\overline{v}_i \mid \exists \overline{\delta} \in V^d : (f1, \overline{\sigma}; \overline{\delta}) \rightarrow^* (f, \overline{v})\} \text{ is a finite set}$$

It is easy to see that in the interpreter of Figure 1, parameters `pgm` and `e` are BSV. The reason: `pgm` is certainly never changed, and `e` is always a substructure of `pgm`, of which there are a finite number.

Note that expressions can be BSV even though dynamically dependent, e.g. `sign(X_D) = if_D X_D > 0 then 1 else 0`.

Infinite static loops. If program `p` is annotated so that every statically annotated parameter or `where` variable is of BSV, then infinite static loops during specialisation can be avoided. A simple-minded strategy that works in principle is to treat *every* program point as a "memoisation point;" [6] the BSV condition ensures that the residual program must be finite in size. A practical strategy yielding a smaller residual program would be for the BTA to mark as "don't unfold" at least one call in every loop that does not properly decrease some static variable. These are easily detected using the parameter dependency graph defined below.

6 Linking Syntax and Semantics

A *call path* of length $k-1$ is a sequence $\pi = [(f^1, \overline{x}), (f^2, \overline{e}^2), \ldots, (f^k, \overline{e}^k)]$, where `p` (assumed tail recursive) contains definitions $f^{i-1}\,\overline{y}$ `=` $\cdots f^i\,\overline{e}^i \cdots$ for $2 \leq i \leq k$. Let $\pi(s,t)$ be π's subpath $[(f^s, \overline{e}^s), \ldots, (f^t, \overline{e}^t)]$ for $1 \leq s < t \leq k$. The *symbolic state transformer for* π, st^π, is an expression defined by $st^\pi = st_k$, where

$$\begin{aligned} st_1 &= \overline{x} \\ st_{i+1} &= [\overline{e}'_1/y_1, \ldots, \overline{e}'_n/y_n](\overline{e}^{i+1}), \text{ where } \overline{e}' = st_i,\ n = \text{arity}(f^i), \\ &\qquad\qquad\qquad\qquad\qquad \text{and } \overline{y} = \text{parameters of } f^i \end{aligned}$$

Intuitively, the state transformer unfolds the calls without doing any computations, expressing the argument tuple for f^k in terms of the variables of f^1.

Extension to non-tail-recursion: account for values returned by function calls.

Definition 2 (Parameter Dependency along a Call Path). We say that f^k_j (the jth parameter of f^k) *depends on* f^1_i *along* $\pi = [(f^1, \overline{x}), \ldots, (f^k, \overline{e}^k)]$ iff expression $(st^\pi)_j$ contains f^1's ith parameter $\overline{x}_i$.

Example 1.
$$p = \begin{cases} \texttt{f(x,y) = if x=0 then 0 else g(x-1)} \\ \texttt{g(u)   = if u=0 then g(u+2) else f(u-2,u-3)} \end{cases}$$

$$\begin{aligned} \pi_1 &= [(\texttt{f}, \texttt{(x,y)}), (\texttt{g}, \texttt{(x-1)}), (\texttt{f}, \texttt{(u-2,u-3)})] \\ \pi_2 &= [(\texttt{f}, \texttt{(x,y)}), (\texttt{g}, \texttt{(x-1)}), (\texttt{g}, \texttt{(u+2)}), (\texttt{g}, \texttt{(u+2)})] \end{aligned}$$

$$st^{\pi_2} = \texttt{(((x-1)+2)+2)} \qquad st^{\pi_1} = \texttt{((x-1)-2,(x-1)-3)}$$

Here $\texttt{f}_1$ depends on $\texttt{f}_1$ but not on $\texttt{f}_2$ along π_1, and $\texttt{g}_1$ depends on $\texttt{f}_1$ along π_2. Note that π_2 is a valid call path, even though no corresponding "real" computation exists (due to the test, `g` cannot call itself twice in a row). In general, *every* real computation is yielded by *some* call path, but not vice versa:

Lemma 3 (Connecting Call Paths and Computations via st^{π}). *For any state transition sequence* $(f^1, \overline{v}^1) \rightarrow (f^2, \overline{v}^2) \rightarrow \cdots \rightarrow (f^k, \overline{v}^k)$ *there exists a call path* $\pi = [(f^1, \overline{x}), (f^2, \overline{e}^2), \ldots, (f^k, \overline{e}^k)]$ *such that* $\overline{v}^t = [\![st^{\pi(s,t)}]\!]_0 \overline{v}^s$ *for all* $1 \leq s < t \leq k$. π *is called* the corresponding call path *of the transition sequence.*

7 Flow Analyses

The purpose of this paper is to find algorithms to automatically discover which parameters are BSV. This is in general undecidable, so the goal is to find a large subset of $\{f_i \mid f$ is a function in p and f_i is of BSV and $1 \leq i \leq \text{arity}(f)\}$.

The analyses work by initially classifying all parameters as being of "unknown binding time $\bot$," and then gradually refining them. Each parameter is classified with one value from the binding time domain: $\bot$ (unknown), D (dynamic), S (static) or B (BSV). When analysis is done, parameters still classified as $\bot$ will be "dead code," and ones classified as S are changed to D ("guilty unless provably innocent").

$$\begin{tikzcd} & & B \arrow[d, no head] \\ D \arrow[dr, no head] & & S \arrow[dl, no head] \\ & \bot & \end{tikzcd}$$

We need information about value flow from one function parameter to the next when evaluating p, so we construct a *parameter dependency graph*:

Definition 4 (PDG). A *parameter dependency graph* is a pair, $\Delta = (V, E)$, with nodes for each parameter and variable in p and one return value node per function:

$$\begin{aligned} V = \ & \{f_i \mid f \text{ is a function in } \mathtt{p} \text{ and } 1 \leq i \leq \text{arity}(f)\} \ \cup \ \{Cst\} \ \cup \\ & \{x \mid x \text{ is a } \mathtt{where}\text{-bound variable}\} \ \cup \ \{f_{\mathbf{R}} \mid f \text{ is a defined function}\}. \end{aligned}$$

The edges of Δ describe the data flow and are constructed by functions *GP* and *GE* below: $E = GP([\![f1\,\mathtt{(}x_1\mathtt{,}\ldots\mathtt{,}x_{m_1}\mathtt{)}\ \mathtt{=}\ e_1]\!])$. We first describe the PDG informally, then give an algorithm for its construction.

Node *Cst* indicates all constant arguments and node $f_{\mathbf{R}}$ indicates the *return value* from a call to function f. Roughly speaking, a construction 'e `where` $x = e'$' with e' containing a call $f(e_1, \ldots, e_m)$ is handled by a flow edge from f's return-value node $f_{\mathbf{R}}$ to the x value node. Further, a call in tail position causes an edge from the callee's return node to the caller's return node.

The PDG for the interpreter is shown in Figure 2; multiple edges with identical labels between node pairs are shown as one edge. The strongly connected component (SCC for short) with multiple nodes is shown with a dashed box. For the last case in the interpreter (the one processing `where`), *GE* generates edges

- for the subexpression '`v4 = Eval(e2, ns, vs, pgm)`': $\mathtt{Eval}_e \rightarrow \mathtt{Eval}_e$, $\mathtt{Eval}_{ns} \rightarrow \mathtt{Eval}_{ns}$, $\mathtt{Eval}_{vs} \rightarrow \mathtt{Eval}_{vs}$, $\mathtt{Eval}_{pgm} \rightarrow \mathtt{Eval}_{pgm}$, $\mathtt{Eval}_{\mathbf{R}} \rightarrow \mathtt{v4}$
- and for the subexpression '`Eval(e1, cons(X, ns), cons(v4, vs), pgm)`': $\mathtt{Eval}_e \overset{\downarrow}{\rightarrow} \mathtt{Eval}_e$, $\mathtt{Eval}_e \overset{\uparrow}{\rightarrow} \mathtt{Eval}_{ns}$, $\mathtt{Eval}_{ns} \overset{\uparrow}{\rightarrow} \mathtt{Eval}_{ns}$, $\mathtt{v4} \overset{\uparrow}{\rightarrow} \mathtt{Eval}_{vs}$, $\mathtt{Eval}_{vs} \overset{\uparrow}{\rightarrow} \mathtt{Eval}_{vs}$ and $\mathtt{Eval}_{pgm} \rightarrow \mathtt{Eval}_{pgm}$

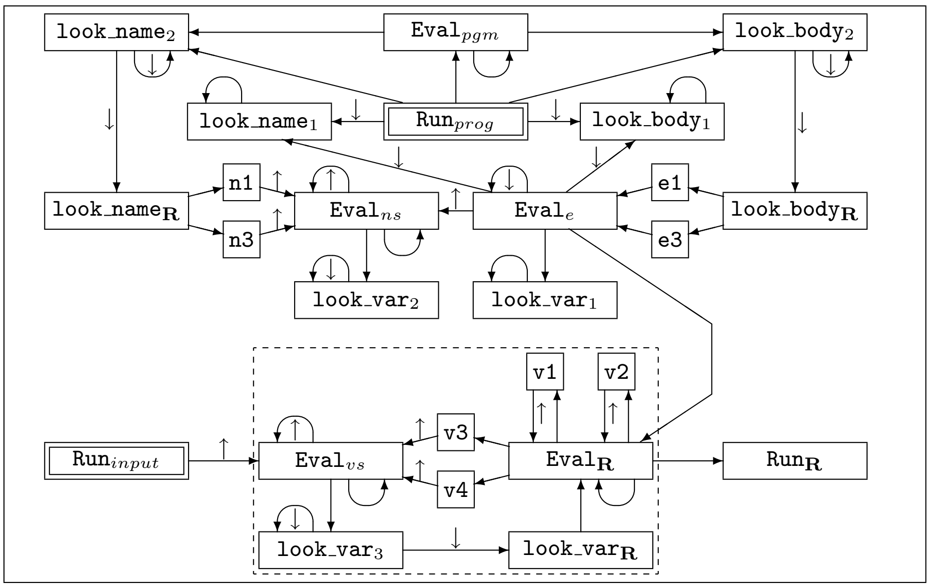

Figure 2: Parameter Dependency Graph Δ for the Simple Interpreter

7.1 Constructing the PDG

The analyses are based on some well-founded partial order $<$ on V ("substructure" in all our examples), but as the analyses work with *expressions* as opposed to *values*, we first define what it means for a relation on expressions to safely describe their value relations.

Define $e_1 \ll e_2$ iff $\forall \overline{v} : [\![e_1]\!]\overline{v} < [\![e_2]\!]\overline{v}$. Then $\prec$ is a *safe size approximation* if $e_1 \prec e_2$ implies $e_1 \ll e_2$; $\preceq$ is defined analogously. An *approximating size transformation description* τ takes values '$\downarrow$' (decreasing), '$=$' (equal) or '$\uparrow$' (increasing) where $\downarrow$ means "definitely less," $=$ means "definitely less or equal" and $\uparrow$ means "no guarantees." If we have not established that $e \preceq e'$ we write $e \not\preceq e'$. In graphs, we will simply omit the $=$ labels to make them less cluttered.

The edge generating functions GP : *FunctionDefinitions* $\rightarrow$ *EdgeSets* and GE : (*Expressions* $\times$ *Nodes* $\times$ *Descriptions*) $\rightarrow$ *EdgeSets* are now defined by $GP(fi\texttt{(...)}\ \texttt{=}\ e_i) = GE(e_i, fi_{\mathbf{R}}, =)$, where

$$GE(e,r,d) = \begin{cases} \{\}, & \text{if } (e \equiv \mathit{constant} \vee e \equiv x) \wedge r = \mathit{nil} \\ \{\mathit{Cst} \rightarrow r\}, & \text{if } e \equiv \mathit{constant} \wedge r \neq \mathit{nil} \\ \{x \xrightarrow{d} r\}, & \text{if } e \equiv x \wedge r \neq \mathit{nil} \\ GE(e_1,r,\uparrow) \cup GE(e_2,r,\uparrow), & \text{if } e \equiv \texttt{cons(}e_1\texttt{,}e_2\texttt{)} \\ GE(e_1,r,\downarrow), & \text{if } e \equiv \texttt{car(}e_1\texttt{)} \vee e \equiv \texttt{cdr(}e_1\texttt{)} \\ GE(e_1,\mathit{nil},\mathit{nil}) \cup & \\ GE(e_2,r,d) \cup GE(e_3,r,d), & \text{if } e \equiv \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\ GE(e_2,x,=) \cup GE(e_1,r,d), & \text{if } e \equiv e_1 \texttt{ where } x \texttt{ = } e_2 \end{cases}$$

and

$$GE(\mathit{fi}\texttt{(}se_1\texttt{,}\ldots\texttt{,}se_{m_i}\texttt{)}, r, d) =$$
$$\begin{cases} \bigcup\limits_{j=1,\ldots,m_i} GE(se_j, \mathit{fi}_j, =) \cup GP(\mathit{fi}\texttt{(...)}\ \texttt{=}e_i) \cup \{\mathit{fi}_{\mathbf{R}} \xrightarrow{d} r\}, & \text{if } r \neq \mathit{nil} \\ \bigcup\limits_{j=1,\ldots,m_i} GE(se_j, \mathit{fi}_j, =) \cup GP(\mathit{fi}\texttt{(...)}\ \texttt{=}e_i), & \text{if } r = \mathit{nil} \end{cases}$$

Base functions other than `cons`, `car`, `cdr` are handled by $\bigcup_{i=1,\ldots,n} GE(e_i, r, d_i)$ for $e = \mathit{basefcn}\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}$, where d_i is $\downarrow$ if $e \prec e_i$, else d_i is blank if $e \preceq e_i$, else d_i is $\uparrow$.

Lemma 5. *Given a PDG $\Delta = (V, E)$ for tail recursive program* `p`*, Δ has a node path $f^1_{i_1} \to f^2_{i_2} \to \cdots \to f^k_{i_k}$ iff there exists a call path π such that $f^k_{i_k}$ depends on $f^1_{i_1}$ along π.*

Extension to non-tail-recursion: omitted for brevity.

7.2 Dynamic and Static Dependency Algorithm

Dynamic parameters are identified by marking all dynamic program input parameters D, then propagating this information along the PDG edges. Dynamic parameters and edges to them *are then removed*, since irrelevant to the remaining algorithms. Next, label all static input parameters and the *Cst* node as S and propagate. This classification ensures that the congruency condition [6] is satisfied. Propagation is done by depth-first search (DFS) in linear time.

7.3 Bounded Domination Algorithm

This algorithm identifies parameters with values always less than or equal to program constants or BSV parameter values. We assume that the PDG Δ has been broken into a DAG (directed acyclic graph) of strongly-connected components (SCCs). It is well known that this can be done in linear time.

Algorithm 6 (Bounded Domination Algorithm, tail recursive case). First re-mark with binding time value B (bounded) the remaining program inputs (none are dynamic) and the *Cst* node. Scan the SCC DAG of Δ from its roots respecting its topological order, and giving priority to any SCC components all of whose nodes are labeled B. For each SCC seen, *if* all in-edges come from nodes marked B *and* there are no edges labeled '$\uparrow$' within the SCC *then* mark all its component nodes B.

Extension to non-tail-recursion: A size description of the return values of calls within `where`s must be computed and used.

Theorem 7. *If a parameter node f is labeled B by the algorithm, it is BSV.*

Δ for Figure 1 is shown in Figure 2. After removing dynamic nodes (all nodes reachable from $\texttt{Run}_{input}$), bounded domination starts with $\texttt{Run}_{prog}$ and marks as B all nodes other than $\texttt{Eval}_{ns}$ and $\texttt{look\_var}_2$. Clearly all these are of BSV

8 Bounded Anchoring

Now we consider variables $\texttt{Eval}_{ns}$ and $\texttt{look\_var}_2$; they are more subtle, as $\texttt{Eval}_{ns}$ can grow. Nonetheless, these *are* BSV even though they get larger when a `where`-expression is interpreted. Intuitively, `ns` is BSV because the number of times `ns` can get larger (without being reset) is bounded by the depth of the nesting of `where`s in `pgm`. This resembles lexicographical ordering: whenever `ns` gets larger in a self-dependent way, `e` gets (strictly) smaller. Of course, `ns` will be "reset" when interpreting function calls, but this does not violate BSV, because the new value is then a substructure of `pgm`.

Given parameters f_i and f_j and a call path $\pi = [(f^1, \bar{e}^1), \ldots, (f^k, \bar{e}^k)]$ where $f^1 = f^k = f$, we say f_i *is anchored in* f_j *along* π iff f_i^k depends only on f_i^1 (itself) or f^1's BSV parameters along π, and $(st^\pi)_j \prec f_j^1$ and BSV(f_j). Anchoring along all call paths is detected by the following reformulation of [4, Theorem 15]. This version works for tail-recursive programs, but needs elaboration to handle nested calls.

Theorem 8 (Bounded Anchoring). *Let C be a SCC of the PDG Δ. If both the following conditions are satisfied, then every f_i in C is BSV:*

1. *If $f_i \in C$ and $g_j \notin C$ and $g_j \to f_i \in \Delta$, then g_j is BSV*
2. *For every $f_i \in C$ and every call path $\pi = [(f^1, \bar{e}^1), \ldots, (f^k, \bar{e}^k)]$ with $f = f^1 = f^k$ where $e_i^k \not\preceq e_i^1$ and $k > 1$ there is a $j(\pi) \neq i$ such that f_i is anchored in $f_{j(\pi)}$ along π (Note: $f_{j(\pi)}$ will be of BSV.)*

In Figure 2, $\texttt{Eval}_{ns}$ is anchored in $\texttt{Eval}_e$ along any path that increases it, and so is of BSV. For another example, suppose the interpreter part implementing "function call":

`'(call f e0) : Eval(e3, list(n3), list(v3), pgm)` were replaced by
`'(call f e0) : Eval(e3, cons(n3, ns), cons(v3, vs), pgm)`,

thus in effect implementing dynamic name scoping. Then $\texttt{Eval}_{ns}$ is anchored in neither $\texttt{Eval}_e$ nor $\texttt{Eval}_{pgm}$, because there is a call path π = [(`Eval`, `(e, ns, vs, pgm)`), (`Eval`, `(e3, cons(n3, ns), cons(v3, vs), pgm)`)], where $(st^\pi)_1 = \texttt{e3} \not\prec \texttt{e} = \bar{x}_1$ (and $\texttt{Eval}_{pgm}$ is unchanged). In actuality, `ns` is now of *unbounded* static variation, thus "dynamic binding" has been detected.

Extension to non-tail-recursion: omitted for brevity.

9 Current, Related, and Future Work

"Bounded Anchoring" is a reworking for efficient partial evaluation of Holst's "quasitermination" test for termination or cyclic nontermination. We emphasize syntactic formulations, efficient graph-based algorithms, independant attribute methods (as opposed to relational), and applications in a real system (Similix). Andersen and Holst [2] are currently working to extend [4] to partial evaluation with higher-order functions.

We are in the process of implementing and evaluating the power and speed of the methods on examples of substantial complexity. The bounded anchoring test is based on "*closed semi-ring*" graph algorithms, see [1]. This is omitted for brevity.

Bounded domination and anchoring may be alternated several times, each iteration possibly detecting new BSV parameters (applying either alone until it stabilises will not suffice). Experiments will be done to find a strategy with good cost-benefit ratio.

The bounded domination and anchoring algorithms can be applied in various orders, after which any remaining S variables should be reclassified as dynamic (it can be seen that this will not violate congruence).

An interesting application is to *discover* lexicographical orderings, e.g. to show that Ackermann's function terminates. The method is in fact more powerful; for example termination of the static parts of Figure 1 seems well beyond current on-line and/or lexicographical methods.

10 A Larger Example

The analyses described in this paper are being implemented, and some first results look promising. The PDG shown in Figure 4 was generated by hand (and is somewhat simplified, e.g. there are no return nodes) from the program text of a small first-order closure-based interpreter (see Figure 3) for the higher-order mini-ML language including simple pattern matching [8]. Static input (a mini-ML program) is passed through $\texttt{run}_e$, and dynamic input (a mini-ML value) is passed through $\texttt{run}_x$.

The sources of BSV are $\texttt{run}_e$ and Cst. For readability, only edges in strongly connected components have been labeled with size information, and edges from Cst are only shown for nodes that have no other in-edges ($\texttt{run1}_{pts}$ and $\texttt{run1}_{vls}$).

When the graph has been generated, the analyses can take place:

Initialisation and Dependency Analyses: All nodes are initially marked $\perp$, $\texttt{run}_x$ is marked D and propagated depth-first to $\texttt{run1}_x$, $\texttt{apfcn}_{arg}$, $\texttt{eval}_{vls}$, $\texttt{apfcn}_{op}$, $\texttt{apbop}_{r_1}$, $\texttt{apbop}_{r_2}$, $\texttt{val-of}_{vls}$ and $\texttt{val-of1}_{val}$. These nodes are then removed. Nodes $\texttt{run}_e$ and Cst are marked S and likewise propagated to the remaining parameters (they are all reachable, so no $\perp$ marks remain).

In the remaining steps only SCCs are considered, in topological order. Nodes $\texttt{run}_e$ and Cst are marked S, and this is propagated to all remaining nodes.

Bounded Domination Analysis: $\texttt{run}_e$ and Cst are marked B. The domination condition is satisfied for $\texttt{run1}_{vls}$, $\texttt{run1}_{pts}$, $\texttt{run1}_e$ and $\texttt{getls}_e$, so B is propagated to them.

Bounded Anchoring Analysis: The remaining SCCs are considered. Node $\texttt{getls}_{pts}$ has an increasing edge, but all loops involving this edge are anchored in BSV sibling loops that decrease $\texttt{getls}_e$, and this is already marked B. Therefore, $\texttt{getls}_{pts}$ is marked B. Node $\texttt{getls}_l$ is treated similarly.

```
run(e, x) ; $Exp \times V \rightarrow V$
  = run1(e, x, (), getls(e, (), ()))

run1(e, x, pts, vls, lms) ; $Exp \times V \times PattList \times VList \times \lambda\ \ List \rightarrow V$
  = apfcn(eval(e, pts, vls, lms), x, lms, lms)

eval(e, pts, vls, lms) ; $Exp \times Patt\ List \times VList \times \lambda\ List \rightarrow V$
  = case e of constant  : constant
     'X                 : val-of(X, pts, vls)
     '(binop e1 e2)     : apbop(binop, v1, v2)
                          where  v1 = eval(e1, pts, vls, lms)
                                 v2 = eval(e2, pts, vls, lms)
     '(if e0 e1 e2)     : if eval(e0, pts, vls, lms)
                          then eval(e1, pts, vls, lms)
                          else eval(e2, pts, vls, lms)
     '(let X=e1 in e2)  : eval(e2, cons('X, pts), cons(v1, vls), lms)
                          where  v1 = eval(e1, pts, vls, lms)
     '(lambda pat body) : list('CLOSURE, list(pts, e), vls)
     '(apply e1 e2)     : apfcn(v1, v2, lms, lms)
                          where  v1 = eval(e1, pts, vls, lms)
                                 v2 = eval(e2, pts, vls, lms)
apfcn(op, arg, l, lms)
  = case (op l) of '((CLOSURE (pts e) vls) ((ps (lambda pat bdy)) lrest))
                        : if (ps (lambda pat bdy))=(pts e)
                          then eval(bdy, cons(pat, ps),
                                    cons(arg, vls), lms)
                          else apfcn(op, arg, lrest, lms)
getls(e, pts, l)
  = case e of constant  : l
     'X                 : l
     '(binop e1 e2)     : getls(e1, pts, getls(e2, pts, l))
     '(if e0 e1 e2)     : getls(e0, pts,
                                getls(e1, pts, getls(e2, pts, l)))
     '(let X=e1 in e2)  : getls(e1, pts, getls(e2, pts, l))
     '(lambda pat body) : getls(body, cons(pat, pts),
                                cons(list(pts, e), l))
     '(apply e1 e2)     : getls(e1, pts, getls(e2, pts, l))
```

Figure 3: Mini-ML Interpreter Text

Reiterating Bounded Domination: Now $\texttt{run1}_{lms}$, $\texttt{apfcn}_{l}$, $\texttt{eval}_{e}$, $\texttt{apbop}_{op}$, $\texttt{val-of}_{i}$, $\texttt{val-of1}_{i}$, and SCC component $\{\texttt{apfcn}_{lms}, \texttt{eval}_{lms}\}$ can be marked B.

Final Iterations: The loops involving the increasing edge from $\texttt{eval}_{pts}$ are now anchored in BSV sibling loops from $\texttt{eval}_{e}$, and can be marked B by bounded anchoring. Now only $\texttt{val-of}_{pts}$ and $\texttt{val-of1}_{pat}$ remain marked as S. Reiteration of bounded domination analysis marks these as B.

Thus we have identified all the static parameters as being BSV, as expected.

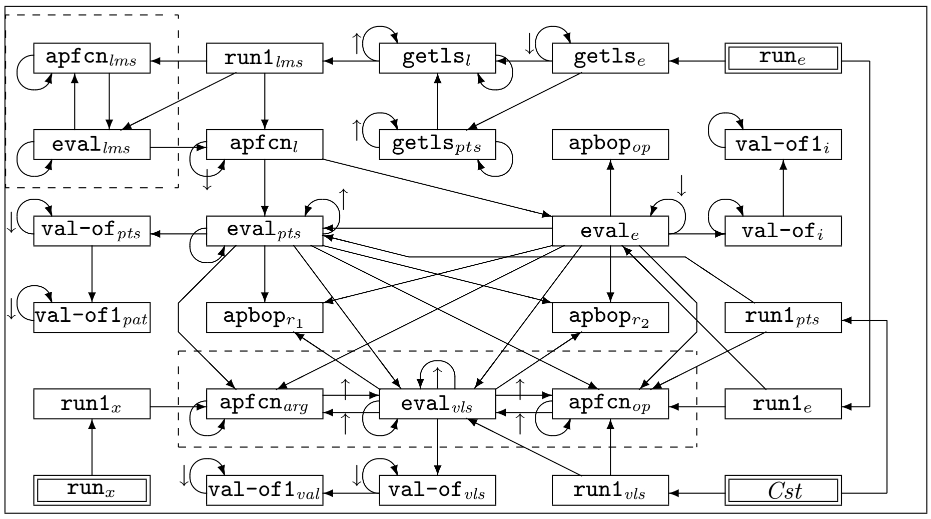

Figure 4: Example Parameter Dependency Graph for the Mini-ML Interpreter

References

1. A.V. Aho, J.E. Hopcroft and J.D. Ullman, 'The design and analysis of computer algorithms' Addison-Wesley, 1975.
2. P.H. Andersen and N.C.K. Holst, Private communication.
3. A.P. Ershov, 'Mixed computation: potential applications and problems for study.' *Theoretical Computer Science*, 18:41–67, 1982.
4. N.C.K. Holst, 'Finiteness analysis,' in J. Hughes (ed.), *FPCA, (Lecture Notes in Computer Science, vol. 523)*, Berlin: Springer-Verlag, 1991.
5. N.D. Jones, 'Automatic program specialization: a re-examination from basic principles,' in D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 225–282, Amsterdam: North-Holland, 1988.
6. N.D. Jones, Carsten Gomard and Peter Sestoft, 'Partial evaluation and Automatic program generation,' Prentice Hall International, June 1993.
7. N.D. Jones, 'What *not* to do when writing an interpreter for specialisation,' in O. Danvy, R. Glück, and P. Thiemann (eds.), *Partial Evaluation. Proceedings*, Springer-Verlag, Lecture Notes in Computer Science, to appear, 1996.
8. G. Kahn, 'Natural semantics,' in F.J. Brandenburg et. al. (eds.), *STACS 87. (Lecture Notes in Computer Science, vol. 247)*.
9. M. Leuschel and B. Martens, 'Global control for partial deduction through characteristic atoms and global trees,' in O. Danvy, R. Glück, and P. Thiemann (eds.), *Partial Evaluation. Proceedings*, Springer-Verlag, Lecture Notes in Computer Science, to appear, 1996.
10. J.W. Lloyd and J.C. Shepherdson, 'Partial evaluation in logic programming,' *Journal of Logic Programming*, 11:217–242, 1991.
11. B. Martens and J. Gallagher, 'Ensuring Global Termination of Partial Deduction While Allowing Flexible Polyvariance,' in L. Sterling (ed.), *ICLP'95, Twelfth International Conference on Logic Programming*, pp. 597–611 MIT Press, 1995